Open Technologies in Formal Verification: Transforming Code for Circuit Validation

Kevin Oliveira
Federal University of Roraima
Boa Vista - RR, Brazil
kevin_costa.tkd@hotmail.com

Herbert Rocha
Federal University of Roraima
Boa Vista - RR, Brazil
herbert.rocha@ufrr.br

Marcelle Urquiza
Federal University of Roraima
Boa Vista - RR, Brazil
marcelle.urquiza@ufrr.br

Francisco Nobre
Federal University of Roraima
Boa Vista - RR, Brazil
diego.nobre@ufrr.br

***Abstract*—This study introduces a methodology for the formal verification of logic circuits described in VHDL, leveraging code transformation and assertion-based property validation. The proposed workflow automatically translates VHDL code into C, enabling the use of advanced analysis tools such as ESBMC, which applies bounded model checking and k-induction techniques. Two translation strategies are presented: a direct approach utilizing a single tool, and a multiple-tool approach that combines distinct translators to improve compatibility with complex VHDL constructs. Code instrumentation ensures that assertions are accurately mapped and verified within the C environment. Experimental evaluation using established benchmarks demonstrates that the multiple-tool approach provides greater flexibility and a higher success rate in both translation and circuit analysis. The results confirm the effectiveness of the methodology in detecting errors and validating safety properties in hardware designs, thereby enhancing the reliability of embedded systems.**

***Keywords*—bounded model checking; VHDL; hardware.**

I. Introduction

Embedded systems (ES) are semiconductor devices with integrated software, designed to connect and interact with other devices. Typically, the primary purpose of an ES is to control and provide information for a specific function [1]. ESs have become pervasive in various aspects of daily life, exhibiting increasing complexity and diversity. This complexity can allow errors to go undetected, potentially resulting in economic losses, product failures, or, more critically, risks to human safety [2]. For instance, as reported in [3], a failure in the electronic circuit board responsible for train speed control led to a collision between two trains at a subway station in São Paulo, according to the Secretary of Transportation.

To achieve high-quality hardware and software systems, it is essential to control their execution and ensure that specified properties are satisfied [4]. For example, prior knowledge of a hardware implementation can enable more efficient usage. In this context, various verification and testing strategies have been researched and applied to ensure the reliability of both software and hardware systems [2], [5]–[7].

Advancements in technology, particularly in electronic design and manufacturing, have significantly increased the importance of hardware in modern society. As circuits become more integrated, faster, and smaller, the complexity of hardware systems has grown substantially. Consequently, late detection of errors can lead to production losses and increased development costs. Hardware verification, therefore, seeks to ensure that circuits meet their design specifications, employing both formal and dynamic techniques [8].

To facilitate the analysis and optimization of digital circuit descriptions, hardware description languages (HDLs) have become widely adopted. Unlike traditional programming languages, HDLs support not only sequential execution but also concurrent and parallel operations. Among these, the VHSIC Hardware Description Language (VHDL) is one of the most prevalent. VHDL enables descriptions at multiple levels of abstraction, with the behavioral level allowing circuits to be defined algorithmically using loops and processes. A key feature of HDLs like VHDL is their support for both sequential and concurrent statements, where declarations remain active and their order is irrelevant [9].

In parallel with the development of HDLs, formal verification has become essential for ensuring the predictability and reliability of critical computational systems. Model checking, a prominent formal verification technique, uses mathematical formalisms to prove properties of reactive systems [10]. This technique exhaustively explores the model's state space to verify whether a given property holds [11], operating fully automatically without user intervention.

The central problem addressed in this work is: **How can the verification of safety properties in VHDL-described logic circuits be improved to map a property into a symbolic reachability problem, determining whether a specific state can be reached from the initial one?**

To contribute to the verification of computational systems, especially in the context of embedded systems, this work aims to apply methodologies and formal verification techniques to programs written in VHDL [12], [13], with a particular focus on Bounded Model Checking (BMC) [4], [14]. The emphasis is on verifying hardware models described at the bit-circuit level in VHDL, utilizing code transformations to generate models with assertions compatible with model checkers such as ESBMC [7], [14] (*Efficient SMT-Based Context-Bounded Model Checker*). The goal is to analyze reachability for error localization and evaluate assertions that represent safety properties.

The main contributions of this work are summarized as follows:

- A novel methodology for hardware verification that streamlines the overall verification workflow and improves the efficiency of validating hardware designs described in VHDL.
- The design and implementation of an automated verification tool for logic circuits in VHDL, which integrates the ESBMC for formal analysis.

II. Hardware Description Languages

Hardware Description Languages (HDLs) were developed to facilitate the design of logic circuits comprising a large number of elements and supporting a wide range of logical and electronic abstractions [15]. Prominent examples include VHDL [16], Verilog [17], and SystemC [18]. As noted by [19], HDLs are specialized programming languages used to model the behavior and structure of digital circuits. These models are used as simulator inputs, allowing thorough analysis and validation of circuit behavior before implementation.

Examples of projects that leverage HDLs include the Rocket Chip [20] generator, which provides open-source implementations of the RISC-V instruction set architecture (ISA). This project utilizes the Chisel hardware construction language to assemble a comprehensive library of generators for cores, caches, and interconnects, culminating in a fully integrated System-on-Chip (SoC). Another prominent example is OpenTitan[1], an open-source root of trust (RoT) project focused on chip security. OpenTitan is managed by lowRISC CIC, a not-for-profit organization committed to collaborative engineering and the long-term development and maintenance of open-source silicon designs and tools.

HDLs provide several key advantages, such as technology- and vendor-independence, which enhance code portability and reusability [9]. Modern HDLs like VHDL and Verilog support both structural and behavioral descriptions, allowing designers to specify not only the interconnection of components but also the intended functionality of the circuit [19]. VHDL is a widely used hardware description language, originally developed in the 1980s to address the requirements of the United States Department of Defense [9].

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity mux2x1 is
port (sel: in STD_LOGIC;
     a,b: in STD_LOGIC;
     y: out STD_LOGIC);
end mux2x1;

architecture dataflow of mux2x1 is begin
y <= (a AND NOT sel) OR (b AND sel);
end dataflow.
```

Fig. 1. Example of a multiplexer described in VHDL.

Figure 1 illustrates a two-input multiplexer described in VHDL. The code begins with library imports, followed by the entity declaration, which defines the module's interface and its input/output ports. The architecture section specifies the functional behavior of the multiplexer, establishing the logical relationship between inputs and outputs as previously declared. The VHDL was selected for this study due to its expressive power and flexibility, which facilitate the implementation and verification of the proposed methodology.

According to [9], a VHDL description can be specified at different levels of abstraction, each serving distinct purposes in the hardware design process:

- **Behavioral Level:** Describes the circuit in terms of its intended algorithmic behavior, using constructs such as loops and processes. This level allows designers to specify what the circuit should do, often employing syntax similar to that of conventional programming languages.
- **Register Transfer Level (RTL):** Models the circuit based on the flow of data between registers and the operations performed on that data by combinational logic. RTL descriptions provide a balance between abstraction and implementation detail, making them suitable for synthesis and optimization.
- **Structural Level:** Specifies the circuit in terms of its actual hardware components and their interconnections, closely reflecting the physical implementation. At this level, designers can define logic gates, modules, and their connections, including timing details such as unit or specific gate delays.

A. Bounded Model Checking

Bounded Model Checking (BMC) is a specialized form of model checking that typically leverages Boolean satisfiability

[1] https://opentitan.org

(SAT) techniques to address the state explosion problem inherent in traditional model checking, which exhaustively explores all possible system states [4]. By limiting the exploration to a finite depth, BMC provides a practical means of verifying properties in complex systems.

In BMC, given a transition system M, a property ϕ, and a bound k, the system is unrolled k times to construct a verification condition (VC) ψ. The VC ψ is satisfiable if and only if there exists a counterexample to ϕ of length less than or equal to k [4], [14]. This approach enables the detection of property violations within a bounded number of steps, making it particularly effective for identifying shallow bugs.

ESBMC [2] is an open-source, context-bounded model checker for C and C++ programs, utilizing satisfiability modulo theories (SMT) to verify both single- and multi-threaded code. Developed collaboratively by the Federal University of Amazonas (Brazil), University of Manchester (UK), University of Southampton (UK), and University of Stellenbosch (South Africa), ESBMC is distributed under a permissive license. Its robust SMT-based approach enables the automated verification of a wide range of safety properties, making it suitable for complex software and hardware systems [7]. ESBMC supports the verification of various safety properties, including arithmetic overflow and underflow, pointer safety, array bounds, and user-defined assertions [4], [14].

In ESBMC, the program under analysis is modeled as a state transition system, represented by the tuple $M = (S, R, S_0)$. Here, S denotes the set of states, $R \subseteq S \times S$ defines the transition relation, and $S_0 \subseteq S$ specifies the set of initial states. Each state $s \in S$ comprises the values of the program counter (PC) and all program variables. The initial state S_0 assigns the entry point of the control flow graph (CFG) to the PC. Transitions are captured by the logical formula $\gamma = (S_i, S_{i+1})$, which encodes the constraints on the PC and program variables between consecutive states [14].

ESBMC was selected for this work due to its comprehensive feature set, including robust SMT-based assertion checking and support for both single-threaded and multi-threaded verification. Furthermore, ESBMC has demonstrated its effectiveness in international competitions such as SV-COMP [3], earning multiple gold, silver, and bronze medals in recent years.

III. Related Work

The work in [21] introduces a C++ tool called `v2c`, which performs code transformation from the Verilog hardware description language to the C programming language. Unlike bit-level translation, `v2c` operates at the word level, improving scalability and enabling the application of advanced verification techniques such as interpolation [22]. This approach would not be feasible with bit-level translations. The tool processes Verilog code by applying semantic rules, mapping operation bits, and generating a word-level C representation known as a software netlist.

In [23], a methodology is proposed that leverages software analysis techniques and tools for circuit analysis, drawing parallels between hardware and software verification approaches. For evaluation, three analysis methodologies are employed using interpolation [22]: abstract interpretation, k-induction [24], and hybrid techniques.

In [25], an approach is presented for timing analysis at the level of individual logic gates within a circuit. The methodology involves translating a VHDL-described logic circuit into a formalism based on timed automata, represented as a finite state automaton with symbolic clocks evolving at a uniform rate. This translation is performed automatically by emulating the propagation of transactions along each signal, effectively modeling the circuit as a network of timed automata. The resulting automata are then analyzed using the UPPAAL tool [26], a model checker specialized in verifying timing properties.

In [27], a methodology is proposed for the dynamic integration of Assertion-Based Verification (ABV) throughout various phases of the embedded systems verification flow, including emulation, diagnostics, and debugging. The authors also introduce a tool called *RadCheck*. The verification process follows the V-model, which divides verification into phases that run in parallel with circuit design stages. At the system level, general functionalities are specified using PSL, while the integration level focuses on interaction problems by defining properties that incrementally cover the system's structural units. At the unit level, internal behaviors are specified through input/output parameters and internal data structures.

A key contribution of the present study is the use of assertions as the primary means of analysis, leveraging ESBMC for their verification. The methodology adapts assertion models from [21], [23], [25], [27] for hardware verification, integrating them into the proposed framework to enhance the effectiveness of property checking in VHDL circuits.

IV. Proposed Method

This work focuses on the analysis of VHDL code through the strategic insertion of assertions, enabling the validation of critical properties and the identification of potential errors during execution. To enhance the verification process, code transformation techniques are employed to translate VHDL into C, leveraging the advanced capabilities of C-based analysis tools. This approach ensures that properties specified in the original hardware description can be systematically checked

[2] https://github.com/esbmc/esbmc
[3] https://sv-comp.sosy-lab.org

using formal verification methods, thereby improving the reliability and robustness of logic circuit designs.

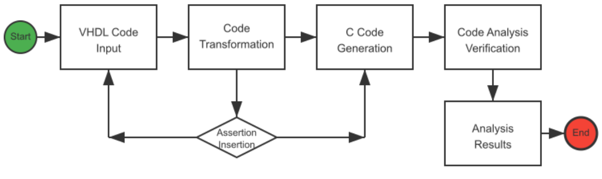

Fig. 2. Flowchart of the proposed method.

As depicted in Figure 2, the process begins with the VHDL code to be analyzed. Notably, property validation is not performed directly on the VHDL code; instead, all analyses are conducted on the code after it has been transformed into C. This approach leverages the availability of mature analysis tools for C, such as ESBMC, which are widely used for error detection and formal verification in this language.

Assertions may be inserted either before or after the code transformation step. Regardless of the chosen approach, it is essential that assertions conform to the syntax and semantics required by the analysis tool to ensure effective verification. The decision regarding the timing of assertion insertion depends primarily on the translation methodology and the capabilities of the translation tool to handle assertions.

If assertions are inserted prior to transformation, they must be translated into C source code automatically by the translation tool, as comments, or through code instrumentation. This instrumentation must preserve the original logic of the code to avoid altering its behavior. Alternatively, if assertions are inserted after the transformation, they can be directly added to the C code, as the code is already in the target language for analysis.

This section presents two translation methods developed in the course of this work. Both methods follow the general structure of the proposed approach, leveraging assertion systems and code transformation tools. To ensure the effectiveness of these approaches, code instrumentation is performed to adapt the source code for compatibility with the tools and to facilitate their execution. The analysis phase, which is common to both methods, will be discussed in Section IV-C, as both approaches utilize the same analysis tool and verification structures. That section will detail how the tool is employed to check code correctness through assertions.

To clarify the steps of the proposed methodology, the code shown in Figure 3 will be used as a running example. This code implements a basic Arithmetic Logic Unit (ALU) concept, consisting only of AND and OR gates, as well as an inverter that conditionally inverts the input signal. The output of the inverter is connected to both the AND and OR gates, whose outputs are then routed to a two-input multiplexer. The final output of the multiplexer is determined by a selection signal, allowing either the AND or OR gate output to be chosen as the ALU result.

```
 library ieee;
 use ieee.std_logic_1164.all;

ENTITY Ula_aor IS
PORT(A,B,Binvert,Op1:IN std_logic;
     Result:OUT std_logic);
END Ula_aor;

ARCHITECTURE Ula_aor_behavl OF Ula_aor IS
SIGNAL and_port: std_logic;
SIGNAL or_port: std_logic;
SIGNAL mux2x1: std_logic;
BEGIN
  PROCESS(A,B,Binvert,Op1)
  BEGIN
    IF(Binvert = ’0’) THEN
      mux2x1 <= B;
    ELSE
      mux2x1 <= NOT B;
    END IF;
    and_port <= A AND mux2x1;
    or_port <= A OR mux2x1;
    IF(Op1 = ’0’) THEN
      Result <= and_port;
    ELSIF(Op1 = ’1’) THEN
      Result <= or_port;
    END IF;
  END PROCESS;
END Ula_aor_behavl;
```

Fig. 3. Example of VHDL code for an ALU with AND and OR gates.

A. Direct Transformation Method

The direct transformation method utilizes a single tool to convert VHDL code into C. In this approach, the logic circuit is initially described in VHDL, and assertions are inserted by the user. The VHDL code, along with these assertions, is then translated into C and instrumented for subsequent verification. The resulting C code is automatically generated and analyzed; if any assertion is violated during analysis, a failure is reported.

A key aspect of this method is the insertion of assertions into the VHDL code. Although VHDL provides its own assertion constructs, a custom assertion model based on comments was developed to support the features required by the chosen model checker, such as the generation of nondeterministic values. As noted in [4], this allows a variable to assume any value within its declared type.

Assertions are placed between the tags @c2vhdl:ASSERT and @c2vhdl:END, with the entire block commented out to prevent errors during translation or synthesis. Each assertion includes the necessary constructs for analysis with the ESBMC tool and consists of three main components: the condition to be checked, an associated message, and the severity level.

```
1  --@c2vhdl:ASSERT
2  --assert (Result = ’1’)
3  --report ”The result was different from 0”
4  --severity ERROR;
5  --@c2vhdl:END
```

Fig. 4. Example of an assertion for verifying the AND gate.

The condition specified on **line 2** of Figure 4 defines the property to be checked by the verification tool. This condition, introduced by `--assert`, may optionally be preceded by the keyword `not`, enabling the user to negate the assertion if required. In the example shown in Figure 4, the assertion verifies whether the variable `Result` is equal to 1. If the value produced by the code does not satisfy the condition stated in the assertion, the verification tool will report a failure.

In addition to assertions, the `__ESBMC_assume()` function is also supported by the ESBMC model checker. This function enables the specification of assumptions about variable values during the verification process. Its primary purpose is to constrain the state space explored by the model checker, allowing the analysis to focus on scenarios where certain input values or conditions are known in advance. This is particularly useful for verifying logic gates or modules where both the input and the expected output are predetermined.

1) Translation of VHDL Code to the C Language: The V2C tool [28] provides several advantages in terms of language translation equivalence; however, it also exhibits notable limitations when translating VHDL code. Specifically, it does not support certain VHDL-specific constructs. The tool only accepts input and output ports of the following types: `bit`, `std_ulogic`, `qsim_state`, `std_ulogic_vector`, and `integer`. Within the architecture section, V2C supports a broader range of constructs, including expressions involving `signal`, `variable`, `integer`, `string`, and character types. For conditional expressions, the supported operators are: `AND`, `OR`, `NOT`, <=, =>, =, <, >, and <>. The tool also recognizes the `process` and `block` structures; however, the `process` construct is limited to `if-else`, `case`, and loop statements. As future work, this study intends to explore new approaches to extend support for VHDL constructs not currently handled by V2C.

During translation, it is necessary to replace the original VHDL operators with their C language equivalents. Exceptions are made for specific operators that manipulate `bit` values, such as concatenation or vector slicing, which require dedicated procedures in C. The translation process generates several arrays to facilitate the mapping of VHDL signals. The `old[]` array stores the values of signals from the previous cycle, which are used for subsequent computations. The `new[]` array holds the values computed during the current cycle, based on the contents of `old[]`. The `chg[]` array contains the result of an exclusive OR operation between the new and old values; if a change is detected, the value from `new[]` is copied to `old[]`.

In the generated C code (see Figure 5), the `in_data[]` and `out_data[]` arrays represent the input and output signals, respectively. The tool reads values from `in_data[]` and writes the processing results to `out_data[]`. At the end of each operation, the final state values stored in the `new[]` array are transferred to `out_data[]`. Any code fragments that are commented out—such as assertions present in the VHDL code—remain unchanged during translation and are utilized in the subsequent verification step.

```
/* Start of Translation */
if (chg[A] || chg[B] ||
  chg[Binvert] || chg[Op1]) {
  __ESBMC_assume(A = ’0’);
  __ESBMC_assume(B = ’1’);
  __ESBMC_assume(Binvert = ’1’);
  __ESBMC_assume(Op1 = ’0’);
  if ((old[Binvert]==0)) {
    if (chg[B]) {
      new[mux2x1]=old[B];
    }
  }
  else {
    if (chg[B]) {
      new[mux2x1]=~old[B];
    }
  }
  ...
}
/* End of Translation */
```

Fig. 5. Example of C code generated by the translation process

2) Code Instrumentation: After translation, the assertions remain commented out in the generated C code. Therefore, code instrumentation is required to enable their support during the analysis phase. All instrumentation steps are performed on the translated C code. The first step is to insert macros at the beginning of the C file, which define the assertion mechanisms to be used. Figure 6 shows the macros implemented for this purpose. In line 1 defines the macro for displaying error messages, while line 2 defines the assertion macro, which invokes the error message macro if the assertion fails. These macros can be used independently of ESBMC in the translated C code.

```
#define log_error(M, ...) \
  fprintf(stderr, M, __FILE__, \
   __LINE__, ##__VA_ARGS__)

#define __MY_assert(A, M, ...) \
  if (!(A)) { \
    log_error(M, ##__VA_ARGS__); \
    assert(A); \
  }
```

Fig. 6. Assertion macros implemented in C.

The next step is to identify the commented assertions within the translated code. Assertions are delimited by the tags `@c2vhdl:ASSERT` and `@c2vhdl:END`, which are used to locate assertion blocks. Upon finding a `@c2vhdl:ASSERT` tag, a loop is executed until the corresponding `@c2vhdl:END` tag is encountered, ensuring that the entire assertion block is processed. The contents between these tags are then passed to a function that extracts the relevant assertion information.

Within each assertion block, the condition, message, and severity are extracted using the tags `--assert`, `--report`, and `--severity`, respectively. This extraction is performed using regular expressions. The extracted information is then inserted into the C code using the previously defined macros. This process is repeated for each assertion found in the code. The handling of the `__ESBMC_assume()` function follows a similar approach.

Additionally, during code instrumentation, nondeterministic input is provided for uninitialized variables or function arguments in the translated C code using the `__VERIFIER_nondet_int()` function. This function is applied to all input variables and signals created within the architecture, ensuring that all necessary variables are properly initialized for verification.

In summary, code instrumentation translates the assertion model used in the VHDL code into the corresponding C model, as well as other functions such as `__ESBMC_assert(new[Result] == 1, "Test")`. After instrumentation, the C code is ready to be used as input for various code verification tools, not limited to ESBMC.

B. Multiple Transformation Method

This approach introduces two transformation tools: the first converts VHDL to Verilog, and the second translates Verilog to C. Initially, the VHDL code is provided to the tool along with an auxiliary file containing the input and output variables, as well as preconditions and postconditions. The code then passes through two translation tools and two instrumentation stages. After translation, the preconditions and postconditions specified in the auxiliary file are inserted into the C code, which is subsequently analyzed. If any condition is violated, a counterexample is presented to the user.

The method begins with the VHDL code to be analyzed, accompanied by an auxiliary file that specifies the code's inputs, outputs, preconditions (conditions that must hold before execution), and postconditions (conditions that must hold after execution). This separation of concerns allows for a clear distinction between the circuit's functional description and its verification requirements.

The auxiliary file, illustrated in Figure 7, must list all input and output variables of the VHDL code, using only the variable names. Input variables are declared on the **INPUT** line, and output variables on the **OUTPUT** line. The subsequent lines define the preconditions and postconditions that will be introduced into the code during later processing steps.

```
INPUT:A==0,B==1,Binvert==1,Op1==0
OUTPUT:Result
PRECONDITION: none
POSTCONDITION:Result == 0
```

Fig. 7. Example of an external configuration file.

The first translates VHDL to Verilog, and the second translates Verilog to C. This two-step translation increases the range of supported constructs and enhances translation flexibility. However, the use of multiple tools can introduce additional sources of translation errors. The first tool, VHD2VL [4], performs the VHDL-to-Verilog translation and is chosen for its superior translation quality. Nevertheless, some limitations must be considered.

For example, VHD2VL supports structures such as `std_logic`, `std_logic_vector`, `integer`, and `boolean`, as well as control constructs like `if`, `elsif`, `else`, and `case`. It also handles clock events and allows module instantiation in VHDL code. However, module instances are ignored during translation, which may lead to incomplete or incorrect analysis. Additionally, any assertions present in the original VHDL code are disregarded by the tool, making it essential to use an external file to specify the assertions and conditions to be checked.

However, the code shown in Figure 8, resulting from the previous step, cannot be directly translated to C, as it does not conform to the standard accepted by the V2C tool [21], which performs the Verilog-to-C translation. Therefore, code instrumentation is required to enable translation. This instrumentation involves modifying variable declarations, adjusting Verilog

[4] https://github.com/ldoolitt/vhd2vl

```
...
module Ula_aor(
input wire A,
input wire B,
input wire Binvert,
input wire Op1,
output reg Result
);
reg and_port;
reg or_port;
reg mux2x1;
  always @(A, B, Binvert, Op1) begin
  if((Binvert == 1'b0)) begin
    mux2x1 <= B;
  end
  else begin
    mux2x1 <= ~B;
  end
  and_port <= A & mux2x1;
  or_port <= A | mux2x1;
  ...
  end
endmodule
```

Fig. 8. Code translated by the Vhd2vl tool.

keywords, and identifying elements within the `always@()` block, which contains the code architecture.

The required modifications include declaring variable names as parameters, followed by their respective types outside the module. Another change concerns the **output reg** declaration: in Verilog, a variable can be declared both as an output and as a register, but for compatibility with the translation tool, these declarations must be separated and placed outside the `always@()` block. Furthermore, no variable should be declared inside the `always@()` block, as this would prevent successful translation. Thus, all variables used during code execution must be declared before the `always@()` block. Additionally, reserved keywords must be handled carefully; for example, Verilog uses the **define** keyword for constant definitions, which may conflict with C syntax. Adjusting these keywords is essential to prevent translation errors.

The second part of the translation is performed by the V2C tool [21]. As previously mentioned, after instrumentation, the Verilog code can be translated to C. Once translated, assertions and variable initializations are inserted into the code. The entire architecture section of the original VHDL code, after translation to C, is encapsulated within a function named after the file. Additionally, a `struct` is defined to contain the variables declared as `output`, as well as any auxiliary variables used during execution, such as signals. Input variables are declared in the `main()` function and passed as parameters to the architecture function.

The use of this `struct` is necessary because these variables may have their values modified during execution. To accurately track state changes, variables are created to store both previous and new values. These variables share the same names as those in the `struct`, with the addition of the `_old` suffix to indicate their previous state. After conversion to C, a new instrumentation step is performed, in which assertions are introduced into the C code. For this, the precondition and postcondition file provided alongside the VHDL code is required.

The `__ESBMC_assume()` function is applied when a variable is initialized in the configuration file; thus, the corresponding initialization is reflected in the generated code. As illustrated in line 1 of Figure 7, variable initializations specified in the configuration file are translated into appropriate assume statements in the C code. This ensures that the specified values are enforced during the verification process.

In summary, code instrumentation in this context consists of inserting the assertions and assumptions defined in the configuration file into the translated C code, as well as incorporating any additional functions required to accurately model the original VHDL behavior. After instrumentation, the resulting C code (see Figure 9) is ready to be analyzed by various verification tools, including but not limited to ESBMC.

C. *Assertion Verification*

The model checker employed in the proposed methodology is ESBMC [7], as detailed in Section II-A. ESBMC accepts C code as input and utilizes SMT solvers for program analysis. For verification, the k-induction technique is primarily adopted. To ensure that the analysis is focused solely on user-defined assertions, the following options are enabled, thereby disabling checks unrelated to the properties of interest:

- `--no-pointer-check`: disables pointer safety checks;
- `--no-div-by-zero-check`: disables division by zero checks;
- `--no-bounds-check`: disables array bounds checks.

By configuring ESBMC in this way, only explicitly inserted assertions are verified, avoiding analysis of unrelated properties. During k-induction, ESBMC systematically checks each assertion while considering predefined conditions and variable values throughout the code. After all unrollings, the tool reports either a successful verification (no property violations detected) or a counterexample (indicating a violation of the specified property), depending on the outcome of the assertion checks.

V. Experimental Evaluation

This section details the planning, execution, and results of the proposed approach. In the first, we focus on evaluating the translation methods, while the second examines the effectiveness of the property verification approach.

```
#include <stdio.h>
#include <assert.h>
#define TRUE 1
#define FALSE 0
struct state_elements_Ula_aor{
_Bool Result;
_Bool and_port;
_Bool or_port;
_Bool mux2x1;
};
void Ula_aor(_Bool A, _Bool B, \
  _Bool Binvert, _Bool Op1, \
  _Bool *Result){
  ...
  __ESBMC_assume(A==0);
  __ESBMC_assume(B==1);
  __ESBMC_assume(Binvert==1);
  __ESBMC_assume(Op1==0);
  if((unsigned char)Binvert == 0){
    sUla_aor.mux2x1 = B;
  }
  else{
    sUla_aor.mux2x1 = !B;
  }
  sUla_aor.and_port = A && mux2x1_old;
  sUla_aor.or_port = A || mux2x1_old;
  ...
  __ESBMC_assert(sUla_aor.Result == 0,”Test”);
}
void main() {
  ...
  A=__VERIFIER_NONDET_BOOL();
  B=__VERIFIER_NONDET_BOOL();
  Binvert=__VERIFIER_NONDET_BOOL();
  Op1=__VERIFIER_NONDET_BOOL();
  Ula_aor(A, B, Binvert, Op1, &Result);
}
```

Fig. 9. C code with assertion insertion.

A. Experimental Evaluation Planning

The evaluation was conducted on a Linux system (Ubuntu) equipped with an Intel Core i7-5500U 2.4GHz processor and 8GB of RAM. The experimental procedure was divided into two main stages. First, the translation methods—namely, the direct transformation and multiple transformation approaches—were tested. The proposed method was initially implemented as a prototype, whose source code is publicly accessible at https://github.com/hbgit/circuit_check To assess the effectiveness and limitations of these translation strategies, the following research questions were formulated:

1) Are the code translation tools capable of handling a variety of VHDL code structures?
2) Can the ESBMC tool successfully analyze all codes translated from VHDL to C?

For this study, a benchmark suite comprising 20 VHDL codes was adopted. These codes were sourced as follows:

- **Author-developed codes:** 10 codes were specifically designed by the author to test the tools. These include basic logic gates, multiplexers, and similar circuits.
- **ISCAS99 benchmark:** 6 codes were selected from the ISCAS99 benchmark suite, which presents a higher level of complexity compared to the author-developed codes. These benchmarks are available at: https://github.com/cad-polito-it/I99T. The selected circuits feature switch-case statements, multiple inputs, and diverse input configurations, thereby increasing their complexity.
- **Vhd2vl examples:** 4 codes were taken from the example set provided with the Vhd2vl tool (https://github.com/ldoolitt/vhd2vl/tree/master/examples). These examples cover a range of complexities and are intended to evaluate both the limitations of the translation tool and the analyzability of the resulting codes. The set includes memory structures, chains of `if` statements, and a counter.

The evaluation focused on the ESBMC analysis tool. This phase was conducted using the translation method that demonstrated the highest success rate, i.e., the approach that successfully translated the largest subset of benchmark codes. The selected translation method was then combined with ESBMC to assess the effectiveness of the verification process. The ESBMC tool is available at https://github.com/esbmc/esbmc.

B. Execution and Analysis of Results

After executing the benchmarks and testing the translation methods, the results are summarized in Table I. The table is organized as follows: the **ID** column uniquely identifies each code; **File Name** specifies the name of the VHDL file; **Approach 01** refers to the multiple transformation method (Section IV-B), with subcolumns for the VHD2VL and V2C tools, as well as the ESBMC analysis; **Approach 02** corresponds to the direct transformation method (Section IV-A), with subcolumns for the V2C tool and ESBMC analysis. For each translation or analysis step, `YES` indicates success, while `NO` denotes failure. A dash (`-`) is used when a subsequent step is not applicable due to a failure in a previous stage.

The ESBMC tool was included in the evaluation to verify whether the generated C code from each translation approach could be successfully analyzed, as ESBMC serves as the post-condition verifier in the proposed methodology. In summary, the multiple transformation approach (VHDL → Verilog → C) demonstrated greater flexibility and compatibility across a diverse set of VHDL designs, establishing it as the preferred method for the proposed verification workflow. In contrast, the direct transformation approach, while effective for simpler cases, is currently constrained by the limitations of available VHDL-to-C translation tools.

TABLE I
RESULTS OF THE TRANSLATION APPROACHES

Official Benchmark						
ID	**File Name**	**Approach 01**			**Approach 02**	
		VHD2VL	**V2C**	**ESBMC**	**V2C**	**ESBMC**
1	AND_ent.vhd	YES	YES	YES	YES	YES
2	XOR_ent.vhd	YES	YES	YES	YES	YES
3	ifchain.vhd	YES	YES	YES	NO	-
4	B01.vhd	YES	YES	YES	YES	NO
5	B06.vhd	YES	YES	YES	YES	NO
6	B10.vhd	YES	YES	YES	YES	NO
7	Comb1.vhd	YES	YES	YES	YES	YES
8	FlipFlop_D.vhd	YES	YES	YES	YES	YES
9	seq1.vhd	YES	YES	YES	YES	YES
10	Ula_tcc.vhd	YES	YES	YES	YES	YES
11	dff.vhd	YES	YES	YES	YES	YES
12	mux_2to1_top.vhd	YES	YES	YES	YES	YES
13	b02.vhd	YES	YES	YES	YES	NO
14	b03.vhd	YES	YES	YES	YES	NO
15	b09.vhd	YES	YES	YES	YES	NO
16	counters.vhd	YES	NO	-	NO	-
17	ifchain2.vhd	YES	YES	NO	NO	-
18	mem.vhd	YES	NO	-	NO	-
19	Nand_ent.vhd	YES	YES	YES	YES	YES
20	Nor_ent.vhd	YES	YES	YES	YES	YES

A detailed comparison of both approaches, based on the benchmark results, is presented below:

Approach 01 (Multiple Transformation):

- All 20 codes were initially processed by the Vhd2vl tool. However, for codes with IDs 16 and 18, the tool failed to correctly link the declared variables, resulting in incomplete translations.
- Of the 20 codes translated to Verilog and instrumented, 18 were successfully converted to C. The failures for codes IDs 16 and 18 were due to the aforementioned variable linking issues. Additionally, code with ID 17 produced invalid output, suggesting that the translator did not properly recognize the Verilog structure during the Verilog-to-C translation.
- Out of the 18 C codes, the 17 were successfully analyzed by ESBMC. The exception was code with ID 17, which, due to translation errors, caused ESBMC to abort the analysis.

Approach 02 (Direct Transformation):

- Of the 20 codes, only 16 were successfully translated by the V2C tool. The primary limitations were related to the use of integer variable declarations and/or the `downto` construct in VHDL, both of which led to translation errors.
- Among the 16 translated and instrumented codes, only 10 were accepted by the ESBMC analysis tool.

These results clearly indicate that the multiple transformation method (Approach 01) outperformed the direct transformation method in terms of translation success and overall compatibility. Notably, this represents a significant improvement over the single-tool translation method previously employed in earlier work. However, it is important to recognize that the use of multiple translation tools—specifically, the combination of VHDL-to-Verilog and Verilog-to-C—introduces additional complexity to the workflow and may increase the likelihood of syntax or semantic errors during translation. In summary, the multiple transformation approach supports more complex VHDL constructs but requires careful tool integration and error handling to ensure reliable verification.

1) Analysis of Code Verification: As previously discussed, the ESBMC results are presented for the approach that demonstrated the best performance in the translation experiments, namely the multiple transformation method. A total of 13 codes were used to evaluate the ESBMC verification tool. Each code includes preconditions (using the `__VERIFIER_assume` function from ESBMC) and postconditions specified as `assert` statements (`__ESBMC_assert()`). The expected outcome for each verification is `TRUE` if the assertion is not violated, and `FALSE` otherwise. Table II summarizes the verification results. The columns include **ID**, **Postcondition**, **Expected Result** (the anticipated outcome based on the pre- and postconditions), and **Execution Time**.

TABLE II
RESULTS OF THE ANALYSIS TOOL

OFFICIAL BENCHMARK			
ID	Postcondition	Expected Result	Execution Time
1	`sAND_ent.f == TRUE`	TRUE	0m1.571s
2	`sAND_ent.f == TRUE`	TRUE	0m0.996s

3	`sb01.outp == 0 &&`
	`sb01.overflw == FALSE`
		TRUE	0m2.127s

4	`sb06.ackout == FALSE &&`
	`sb06.enable_count == FALSE`
		TRUE	0m2.017s

5	`h == a^!(b && c)`	FALSE	0m1.367s
6	`sFlipFlop_D.Q == D`	FALSE	0m1.050s
7	`sseq1.out == TRUE`	FALSE	0m1.637s
8	`sUla_tcc.Resultado == 0`	TRUE	0m3.284s
9	`sdff.data_out == FALSE`	TRUE	0m2.417s
10	`smux_2to1_top.f == TRUE`	TRUE	0m2.334s
11	`sb02.u == TRUE`	FALSE	0m2.086s
12	`sNAND_ent.f == TRUE`	FALSE	0m1.106s
13	`sAND_ent.f == TRUE`	FALSE	0m0.774s

It is important to highlight the postconditions presented in Table II, under the Postcondition column. For IDs 1, 2, 12, and 13, the postcondition `sAND_ent.f == TRUE` ensures the integrity of logical operations and prevents data corruption, thereby preserving the structural correctness of the circuit. In ID 3, the property verifies that, in the absence of overflow, the output remains in a safe state. For ID 4, the assertion guarantees that the acknowledgment and count enable signals do not occur simultaneously in the interrupt handler, preventing potential race conditions. In ID 6, the property checks for correct behavior on clock edges, ensuring that states are maintained across transitions.

Overall, the ESBMC tool successfully identified and evaluated all assertions, yielding results consistent with the expected outcomes. Notably, in cases where assertion violations were anticipated, ESBMC correctly reported errors and provided counterexamples, demonstrating its effectiveness in detecting both valid and invalid property conditions. This confirms the tool's reliability in verifying complex safety properties and its suitability for rigorous hardware verification workflows.

VI. Conclusion and Future Works

This study presented a methodology for analyzing logic circuits described in VHDL by applying code transformation techniques and model checking. The primary goal was to systematically explore the reachable states of a circuit to identify errors that could lead to system malfunctions. The proposed approach introduced a new workflow for transforming VHDL code into C. The multiple transformation approach achieved 65% successfully translated and accepted by the analysis tool—despite requiring the integration of two separate tools. For future work, it is essential to investigate into auxiliary tools will help to better delineate the method's limitations and enabling fully automated property-based testing, further streamlining the verification process for developers.

References

[1] U. B. K. Ramesh, S. Sentilles, and I. Crnkovic, "Energy management in embedded systems: Towards a taxonomy," in *Proceedings of the First International Workshop on Green and Sustainable Software*. IEEE Press, 2012.

[2] G. Cabodi, C. Loiacono, M. Palena, P. Pasini, D. Patti, S. Quer, D. Vendraminetto, A. Biere, and K. Heljanko, "Hardware model checking competition 2014: an analysis and comparison of solvers and benchmarks," *Journal on Satisfiability, Boolean Modeling and Computation*, 2016.

[3] G1. (2012) Acidente foi provocado por falha em circuito eletrônico, diz secretário. [Online]. Available: http://g1.globo.com/sao-paulo/noticia/2012/05/acidente-foi-provocado-por-falha-em-circuito-eletronico-diz-secretario.html

[4] H. Rocha, H. Ismail, L. Cordeiro, and R. Barreto, "Model checking embedded c software using k-induction and invariants (extended version)," *arXiv preprint arXiv:1509.02471*, 2015.

[5] H. Rocha, L. Cordeiro, R. Barreto, and J. Netto, "Exploiting safety properties in bounded model checking for test cases generation of c programs," *4th Brazilian Workshop on Systematic and Automated Software Testing (SAST)*, 2010.

[6] F. Muttenthaler, S. Wilker, and T. Sauter, "Lean automated hardware/software integration test strategy for embedded systems," in *22nd IEEE International Conference on Industrial Technology (ICIT)*, 2021.

[7] R. S. Menezes, M. Aldughaim, B. Farias, X. Li, E. Manino, F. Shmarov, K. Song, F. Brauße, M. R. Gadelha, N. Tihanyi, K. Korovin, and L. C. Cordeiro, "Esbmc v7.4: Harnessing the power of intervals: (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2024.

[8] M. Boule and Z. Zilic, "Efficient automata-based assertion-checker synthesis of seres for hardware emulation," in *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*. IEEE Computer Society, 2007.

[9] D. Cappelatti, *Praticando VHDL*. Editora Feevale, 2010.

[10] I. Grobelna, "Formal verification of control modules in cyber-physical systems," *Sensors*, 2020.

[11] C. Baier, J.-P. Katoen, and K. G. Larsen, *Principles of model checking*. MIT press, 2008.

[12] A. Biere, "The aiger and-inverter graph (aig) format," 2016. [Online]. Available: Availableathttp://fmv.jku.at/aiger

[13] S. Gupta, A. John, and M. Kalra, "Assertion based verification using yosys: A case study from nuclear domain," in *Proceedings of the 16th Innovations in Software Engineering Conference*. Association for Computing Machinery, 2023.

[14] L. Cordeiro, B. Fischer, and J. Marques-Silva, "Smt-based bounded model checking for embedded ansi-c software," *IEEE Transactions on Software Engineering*, 2012.

[15] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*. Springer US, 2008.

[16] IEEE, "Iec/ieee international standard - behavioural languages - part 1-1: Vhdl language reference manual," *IEC 61691-1-1:2011(E) IEEE Std 1076-2008*, 2011.

[17] ——, "Ieee standard for verilog hardware description language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.

[18] ——, "Ieee standard for standard systemc language reference manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, 2012.

[19] E. Christen and K. Bakalar, "Vhdl-ams-a hardware description language for analog and mixed-signal applications," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 1999.

[20] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," Tech. Rep., Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[21] R. Mukherjee, M. Tautschnig, and D. Kroening, "v2c – a verilog to c translator," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2016.

[22] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *International Conference on Computer Aided Verification*. Springer, 2011.

[23] R. Mukherjee, P. Schrammel, D. Kroening, and T. Melham, "Unbounded safety verification for hardware using software analyzers," in *Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016.

[24] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer, "Software verification using k-induction," in *International Static Analysis Symposium*. Springer, 2011.

[25] A. Bara, P. Bazargan-Sabet, R. Chevallier, D. Ledu, E. Encrenaz, and P. Renault, "Formal verification of timed vhdl programs," in *Forum on Specification & Design Languages (FDL)*. IET, 2010.

[26] A. David, P. G. Jensen, K. G. Larsen, M. Mikučionis, and J. H. Taankvist, "Uppaal stratego," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2015.

[27] G. Di Guglielmo, L. Di Guglielmo, F. Fummi, and G. Pravadelli, "On the use of assertions for embedded-software dynamic verification," in *Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE, 2012.

[28] A. Gatti and C. Ghezzi, "Vhdl 2 c: Studio e realizzazione di una tecnica di traduzione automatica del vhdl in linguagio c," http://web.tiscali.it/sito01/pro/v2c/main.htm, Julho 1995.